RL-TR-97-108
Final Technical Report
October 1997

A MULTIPROCESSOR PROTOTYPE FOR ADVANCED SIGNAL PROCESSING

State University of New York, Binghamton

Kanad Ghose

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-108 has been reviewed and is approved for publication.

APPROVED: Darren M. Haddad

DARREN M. HADDAD
Project Engineer

FOR THE DIRECTOR:

JOSEPH CAMERA, Technical Director
Intelligence & Reconnaissance Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/IRAA, 32 Hangar Road, Rome, NY 13441-4114. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1997	Final Sep 94 - Sep 95

4. TITLE AND SUBTITLE

A MULTIPROCESSOR PROTOTYPE FOR ADVANCED SIGNAL PROCESSING

5. FUNDING NUMBERS

C - F30602-93-C-0229
PE - 63260F
PR - 3481
TA - 00
WU - P9

6. AUTHOR(S)

Kanad Ghose

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Department of Computer Science
State University of New York
Binghamton, NY 13902-6000

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/IRAA
32 Hangar Road
Rome, NY 13441-4114

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-108

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Darren M. Haddad/IRAA/(315) 330-2906

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The main goal of this project was to construct a prototype DSP multiprocessing system based on the Texas Instrument TMS320C40 multiprocessing-capable CPUs and show that it provided adequate performance on the Rome Laboratory (RL) Speaker Identification (#Spkrid") code and the Speech Enhancement (#SEU") code. The original proposal called for a 8-CPU prototype and the funded project was for constructing a 4-CPU prototype.

The prototype system that was constructed showed that the computational aspects of the Spkrid code and the SEU code can be easily met on the 4-CPU prototype. (At least one additional CPU, not available in the current prototype, is needed for interfacing to the A/D, D/A components.) The bulk of the proposed effort was involved with the parallelization and porting of the supplied code to the target system, and instrumenting the system to get performance measures that will indicate how the system will perform in real-time.

14. SUBJECT TERMS

A prototype DSP multiprocessing system

15. NUMBER OF PAGES

36

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

1. OBJECTIVES

The main goal of this project was to construct a prototype DSP multiprocessing system based on the Texas Instrument TMS320C40 multiprocessing-capable CPUs and show that it provided adequate performance on the Rome Laboratory (RL) Speaker Identification ("Spkrid") code and the Speech Enhancement ("SEU") code. The original proposal called for a 8-CPU prototype and the funded project was for constructing a 4-CPU prototype.

The prototype system that was constructed showed that the computational aspects of the Spkrid code and the SEU code can be easily met on the 4-CPU prototype. (At least one additional CPU, not available in the current prototype, is needed for interfacing to the A/D, D/A components.) The bulk of the proposed effort was involved with the parallelization and porting of the supplied code to the target system, and instrumenting the system to get performance measures that will indicate how the system will perform in real-time.

2. HARDWARE CONFIGURATION OF THE PROTOTYPE SYSTEM

The prototype DSP system was based on a development system using four TMS320C40 DSP CPUs running at 32 MHz. The characteristics of the prototype system, shown in Figure 1, are as follows:

- Each CPU has an on-chip RAM of 1 K words, each word being 4 bytes. The program activation stack is usually implemented entirely within this RAM, since accesses to this RAM is faster than accesses to external RAMs. Frequently used variables that cannot be allocated within the CPU registers are also kept within the on-chip RAM.

- Each CPU has its own private RAM of 16 K words, each word being 4 bytes. This is in addition to the 1 K word of on-chip RAM within each CPU. Variables local to a CPU are generally allocated in this private RAM. A CPU accesses its private external RAM using its own set of address/data lines.

- Each CPU also has access to a global shared RAM of 128 K words. Global variables, that are shared among the CPUs are usually mapped onto the global RAM. A CPU accesses the global RAM using a shared bus - there is thus bus contention and associated delays in accessing the global RAM.

- Each CPU has an 8-bit wide message passing connection to the other three CPUs.

- Each CPU also has an 8-bit wide message-passing link to off-board CPUs or other components, such as A/D interfaces.

- The Host system is a Sun SPARCLX, running SunOS 4.1.3, and is interfaced to the prototype DSP system through an in-circuit emulator, which communicates with the board via a serial (and relatively slow) JTAG cable.

The C-based compiler and development environment was run on the host. Programs were developed entirely in C on the host, using the message-passing library routines. The compiled programs were downloaded onto the 4-CPU prototype and run on the prototype. I/O from the host was confined between the DSPs and the global memory, since the slow data rate on the JTAG cable precluded I/O at a faster rate.

An attempt was made to interface an Ariel A/D, D/A unit to the prototype system by using one of the free 8-bit wide communication port from a CPU. This involved writing drivers in the assembly language of the

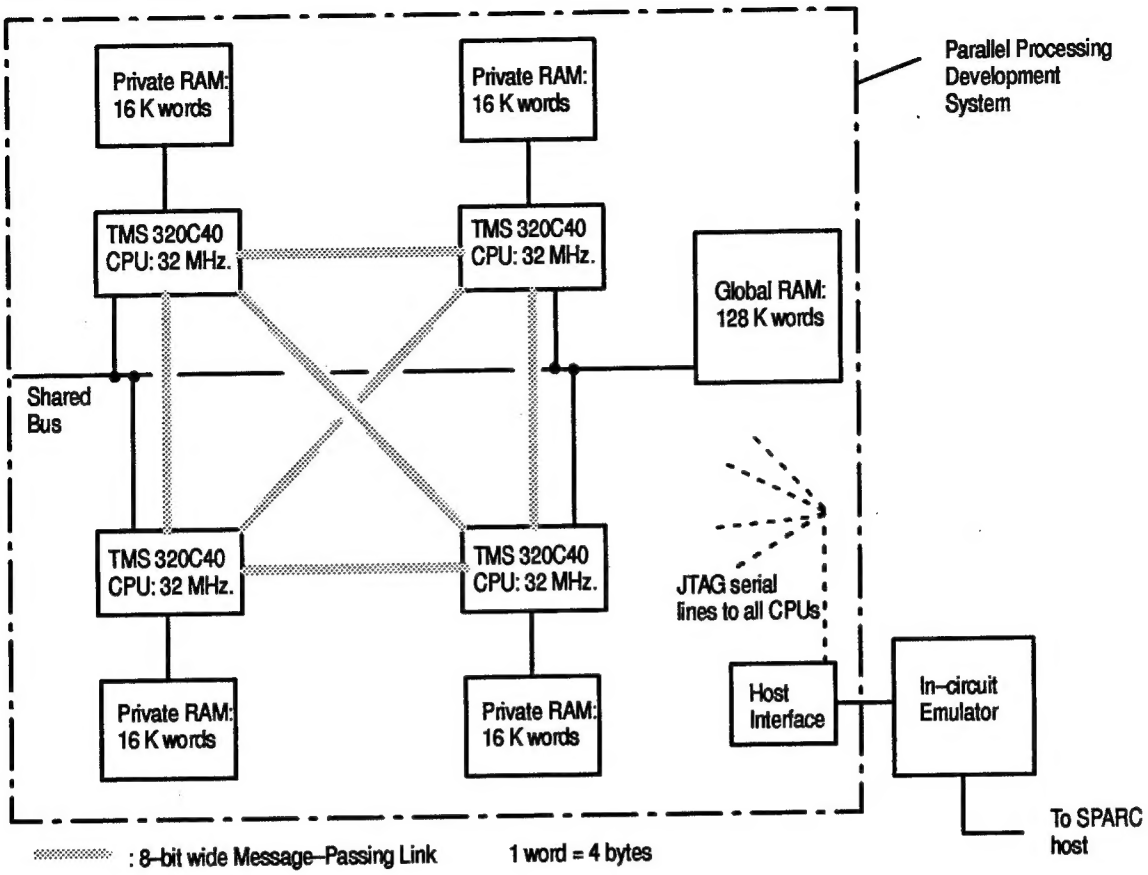


Figure 1. The Hardware Configuration of the Prototype System

TMS320C40 to allow signals and protocols for the Ariel interface to be followed. The resulting interface, however, hung often due to noise and signal problems in the wiring and the communication ports, which were beyond our control. Interfacing circuits were also tried using FPGA based devices, but suffered from the same problems. The real solution is to use a A/D, D/A card designed specifically for the TMS320C40 systems, which was not available to us.

An alternative solution is to use the SPARC host to interface with the Ariel unit and do all signal I/O through the SPARC host. This approach was implemented, but the resulting I/O transfer rate between the host and the prototype was unacceptably slow due to the low data rate through the serial JTAG interface.

3. THE IMPLEMENTATION OF THE SPEAKER IDENTIFICATION CODE ON THE PROTOTYPE SYSTEM

This section describes our approach to the porting and parallelization of the RL Speaker Identification program and an evaluation of the performance of the code on the 4–CPU prototype system. In particular,

we describe the important changes made to the code to facilitate the implementation on the prototype without compromising any functionality. The performance results reported here also identify the main cycle sinks in the program in an effort to show where parallelization efforts should be directed if additional CPUs are available for further performance enhancements.

3.1 The Rome Lab Speaker Identification Program: An Overview

The RL speaker identification ("spkrid") program uses a two-phased approach to recognize speakers. The first phase involves "training" the system with samples from known speakers to set up a database of the features of each speaker. The second phase involves the acquisition of sample data from an unknown source, extracting its feature and comparing the extracted features against the features stored in the database to find a match with one of the known speakers. Currently, this recognition phase does not run in real-time.

Figure 1 depicts the main steps involved in the training phase. The **lbg_vq** algorithm is used to classify the

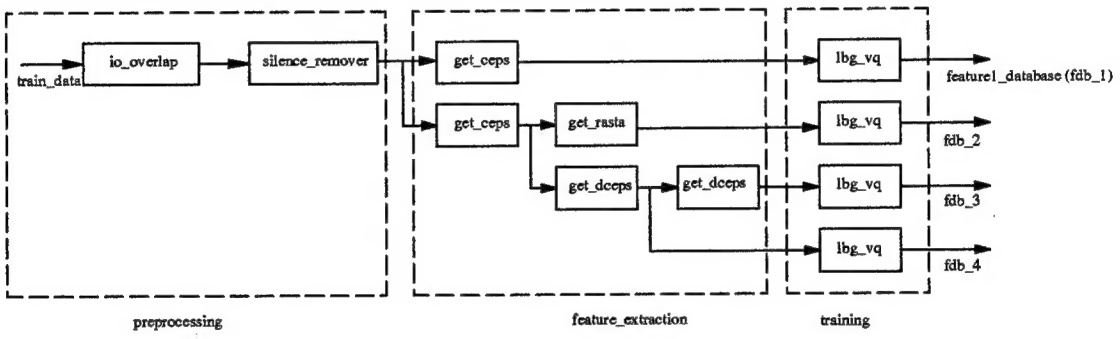


Figure 2. The training phase of the spkrid program

features of the speakers. Four feature files are created for each speaker and the data for all speakers is collected to form a database on a per feature basis. Disk I/O is involved heavily in this phase, since the extracted features are stored in several files on the disk.

Figure 2 depicts the main steps in the recognition phase. Note that the **preprocessing** and **feature_extraction** steps are same as that of training phase. The **feature_extraction** generates data about the four features of the input. For every feature the **classify** module uses the corresponding feature database to come up with the most likely identity of the unknown speaker. The **report** module then uses a statistical maximum likelihood estimation technique to identify the speaker. Again, disk I/O is required to access the stored feature files and for storing the results of the match.

3.2 Estimate Of Operational Complexity For The Speaker ID Program

To understand the complexity of the Speaker ID program and identify code sections that need to be parallelized to meet the throughput requirements, we looked at each of the main modules of the program and estimated the operation counts. The following flowchart shows the data flow among the main modules of the Speaker ID program:

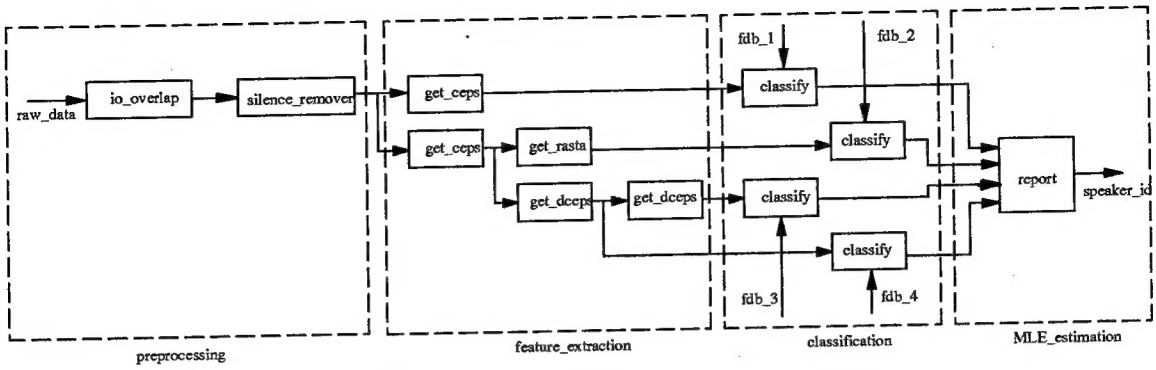
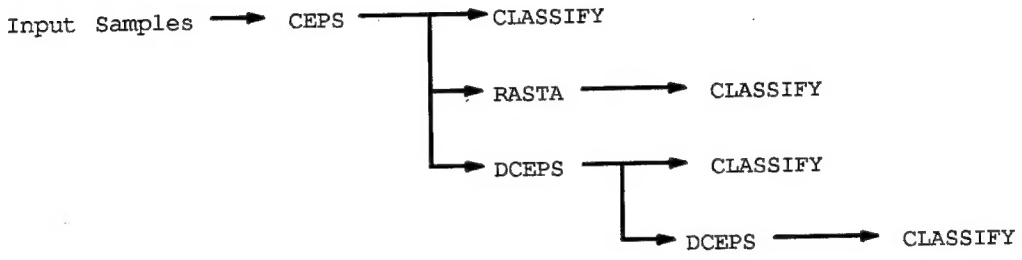


Figure 3. The recognition phase of the spkrid program



I. Notations:

The following notations are used in the complexity analysis.

ALU = ALU operation (*, +, / etc. – mostly floating point ops)
MEM = memory operation (load or store)

We also assume that the overall performance is dictated mainly by the ALU and MEM operations. Thus, register–to–register moves and other similar operations are ignored since they do not dominate the code.

II. Opcounts on a module by module basis:

The CEPS module takes M input samples, and outputs N sets of coefficients; N = M / W (W = window width), each set contains Q coefficients (Q = order). Current values used are: W = 256, Q = 14.

The RASTA and DCEPS modules read in coefficients, and output the same number of coefficients.

Operations per module:

CEPS: 35 ALU, 20 MEM -- per input sample
RASTA: 9 ALU, 11 MEM -- per input coefficient
DCEPS: 42 ALU, 23 MEM -- per input coefficient
CLASSIFY: 3*C ALU, 2*C MEM -- per input coefficient, per speaker, where C = codebook size (currently 40)

Number of ops for each set of input/output coefficients:

CEPS: 8960 ALU, 5120 MEM
RASTA: 126 ALU, 154 MEM
DCEPS: 588 ALU, 322 MEM
CLASSIFY: 1680 ALU, 1120 MEM (per speaker)

III. Total ALU and memory operations:

Approximated formula for total number of operations (assuming current values for parameters):

ALU = M * (40 + 7 * num_speakers)
MEM = M * (23 + 5 * num_speakers)

where M is the number of input samples

The above analysis clearly shows that CEPS and CLASSIFY routines are most compute-intensive, as well as memory-intensive. Parallelization at a fine grain should thus first target these modules.

3.3 Some Possible Approaches For Parallelizing The Speaker ID Program

In this section, we present various approaches for parallelizing the RL spkrid program. In a later section, we briefly discuss these schemes to identify the approaches that are best suited for implementation on the TMS 320C40 based speech multiprocessor prototype.

3.3.1 Application level parallelization

This is the highest level of parallelism where, one processor is allocated to handle one data channel. Full copies of the application run independently on each processor and there is almost no necessity of inter processor communication. This approach is suitable if the number of channels monitored and the number of processors almost match. This approach does not improve the computation time of individual channels, but increases throughput of the system by simultaneously processing more than one channel. If the system offers more processors than channels, clearly, this approach makes poor (zero) utilization of the spare processors. Also I/O traffic may be high as all processors demand access to all the database files. Figure 4 summarizes this approach.

3.3.2 Pipelined computation

Figure 5 depicts another approach for parallelizing the spkrid program based on the use of pipelining to overlap the execution of component steps of the program. In this approach a pipeline of processors is setup to perform partial computation on the data and feed the output to the next processor in the pipeline. Figure 5 shows the use of a 4 stage pipeline, each processor P_i computing only a specific module out of the complete computation sequence. Every stage of the pipeline must be able to process input data when it is available and generate useful output for the subsequent stage before the input is fully available. This scheme is useful when the spkrid program has to process input samples for recognition on a continuous basis. In such a case, the computation overlap is across the unique steps required to process each input sample.

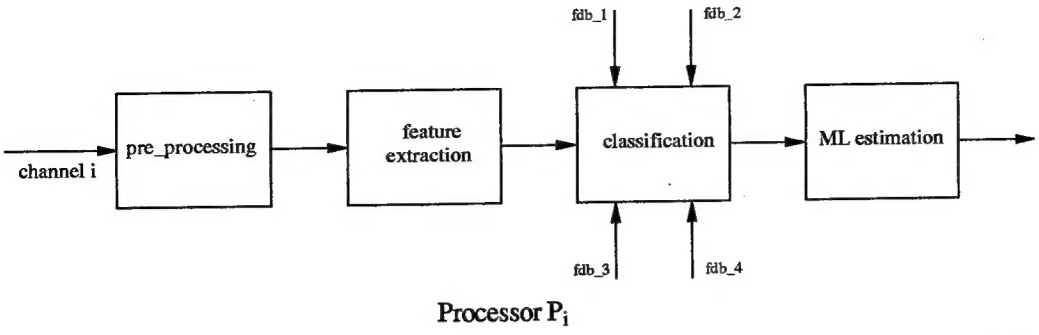


Figure 4. Coarse-grained application-level parallelization of the spkrid code

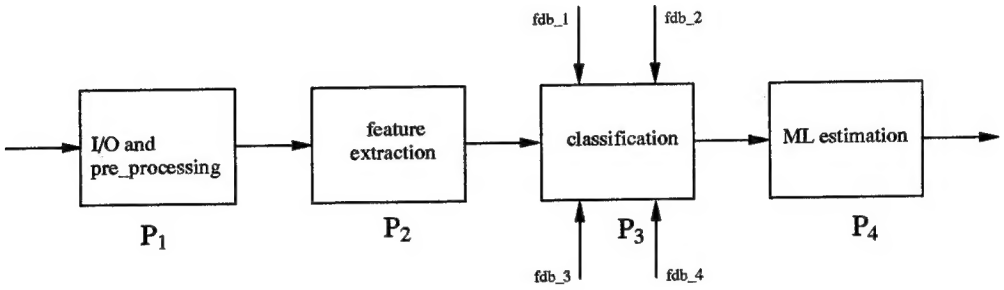


Figure 5. Pipelined implementation of the spkrid code

Another use of the pipelined scheme will be to "simultaneously" run the spkrid program for different channels - in this case the pipeline overlap is among the steps for the various channels. If spare processors are still available, several such pipelines can be setup to increase utilization. The channels can then be suitably distributed and multiplexed based on the number of pipelines available.

With the pipelined approach, the computation time of each pipeline stage must be approximately equal to avoid bottlenecks. However, the above breakup may not be useful since some stages such as classification, require a disproportionately longer execution time. This stage may need to be broken up further, or else duplicated, so that it does not impose a throughput bottleneck.

3.3.3 Hybrid pipelining

In the pipelined approach, the preprocessing step and, perhaps, the feature extraction steps, have a smaller computation time compared to the other steps. If multiple pipelines are used, one for each channel, a common preprocessing (and/or feature extraction) stage(s) can be multiplexed across the channels. This approach, which we call *hybrid pipelining* is depicted in Figure 6. This approach helps to even out the computation time of the stages. Note however, that all the classification stages still demand access to all the feature database files. This can be a problem if a single disk device is present.

3.3.4 Hybrid pipelining with independent feature classification

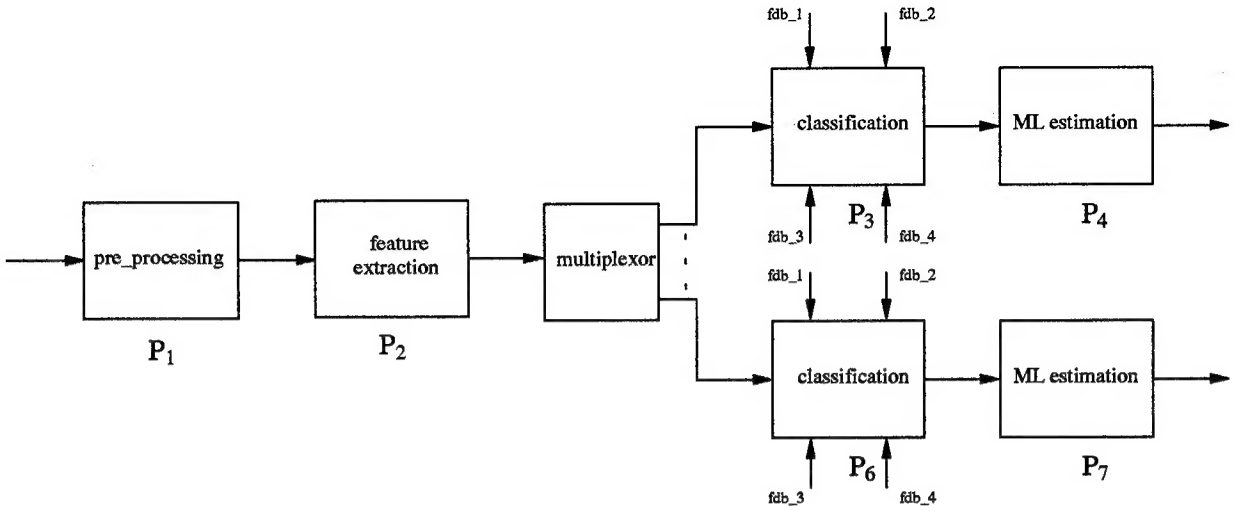


Figure 6. Hybrid, multipipelined implementation of the spkrid code

This technique allocates one processor per feature classification. This approach is depicted in Figure 7. The I/O traffic can be eased due to this separation. Now each processor needs access to only a specific feature database to perform the classification.

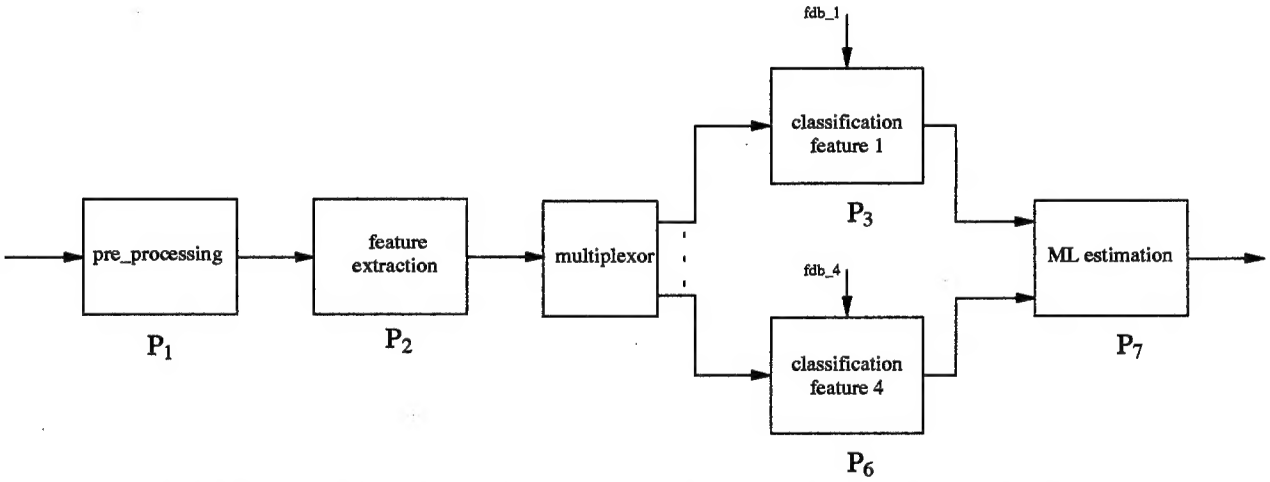


Figure 7. Hybrid pipelined implementation of the spkrid code with independent classifiers

3.3.5 Code level parallelism

This kind of parallelism is achieved by developing parallel code specific to the system architecture. This is a low level of parallelism, popularly known as fine grain parallelism that seeks out concurrent operations at the level of individual code fragments, statements and expressions. The operational complexity analysis reported in Section 3.2 seems to suggest that this is possibly the best way to speed up the CEPS and CLASSIFY routines for a system that allows an adequate number of CPUs. The "adequate number" in this case has to be significantly higher than the four CPUs available on the prototype DSP multiprocessor, since

fine-grain parallelism involves a higher degree of concurrent activity compared to any of the techniques we have discussed thus far.

Fine-grained parallelism can be exploited to a great extent in the following constructs (among others) in a typical program:

- Loops
- Nested Loops
- Recurrence relations

Program loops thus provide a good source of low-level parallelism. Speech processing applications have a profusion of loops in the code. Such loops can be broken into multiple smaller loops, and each loop can be computed on a different processor. Nested loop computation can also be speeded up to a great extent. Recurrence expressions can be parallelized by factoring out constants from the expression. The *classification* module may benefit from this approach. We give below, examples of parallelizing loops and recurrence expressions occurring in the classification code. Data dependency analysis and knowledge of the processor interconnection topology is very essential to generate optimized parallel code. Processor synchronization must also be ensured for correct computation. An example of code level parallelization is shown below.

```
float find_mse(double codeword[],double data[],int vect_len)
{
            int i;
            double;

            mse = 0.0;
            for (i=1; i<=vect_len; i++)
                        mse += codeword[i]-data[i]*codeword[i]-data[i];
            mse /= vect_len;
}
```

This code can be parallelized as follows. Suppose there are $N=2^n$ processors. Then firstly, each processor computes a local value of mse, on data vectors vect_len/N long. Then these mse's are summed up in n steps as follows: In the 1st step processors 0, 2, 4, ..., receive values from 1, 3, 5, ..., and add them locally. In the 2nd step 0, 4, 8, ..., receive values from 2, 6, 12, ..., and so on. In the n^{th} step, 0 receives value from 2^{n-1} to compute the final value of mse.

As another example of fine-grained parallelization, consider the following code:

```
void lpc_cepstrum (double pe, double a[],int p,double ce[])
{
            int i, j;
            ce[0] = log(pe);
            ce[1] = -a[1];          /* all others are 0.0 */
            for (i=2; i <=p; i++)
```

```
        {
                for (j=1; j<ii; j++)
                        ce[i] -= (i-j)*ce[i-j]*a[j];
                ce[i] /= i;
                ce[i] -= a[i];
        }
}
```

This computation can be mathematically represented as

$$ce[i] = -\frac{1}{i}\left(\sum_{j=1}^{i-1} j * ce[j] * a[i-j]\right) - a[i]$$

$$ce[i] = -\frac{1}{i}\left((i-1) * a[1] * a[i-1] + \sum_{j=1}^{i-2} j * ce[j] * (a[i-j] - a[1] * a[i-1-j])\right) - a[i]$$

Notice that in the second equation, the computation of ce[i] is independent of ce[i–1]. Thus we have improved the minimum dependency distance from 1 to 2, and as a result consecutive ce*s* can be computed in parallel on 2 processors. Further parallelism can be extracted by unrolling the loop.

3.4 Parallelization for the TMS 320C40–based Prototype System

The parallelization techniques presented in Section 3.3 are representative of the techniques that can be used to parallelize the spkrid program. However, not all of these techniques may be suited for an efficient implementation on the TMS320C40 based multiprocessor prototype using four CPUs.

Since the 320C40 is primarily a message passing system, parallel implementations at a medium to coarse grain may be preferred. Also, since the number of communication ports on the TMS320C40 are limited, one has to be concerned about the resulting topology. The use of a pipelined approach or the hybrid pipelined approach with independent feature extraction may well be preferred, since it demands a simple interconnection topology. This is, however, not to say that fine–grained approaches cannot be supported at all – the external shared memory in the TMS320C40 can be used for inter processor communication for inter–CPU communication for the fine–grained parallelized version of the spkrid program.

In general, for speaker identification in real–time, disk I/O can be avoided and thus disk I/O bottlenecks are absent. This is because the extracted feature files are typically small in size – small enough to fit inside the 4K internal RAM of each TMS320C40 that performs the classification.

The availability of only four CPUs on the prototype DSP multiprocessor, together with memory size and connectivity limitations drastically affected our approach to the parallelization of the spkrid program. First, and obviously, we had to restrict the degree of parallelism to four. This rules out the possibility of using balanced simple and hybrid pipelined structures, where each pipeline stage performed roughly the same amount of computation as the other stages. (Recall that the CEP and CLASSIFY modules perform a significantly larger number of computation compared to the other modules. To balance the activities among the pipeline stages, each of these modules will have to be broken down into several stages, clearly requiring more than 4 stages overall and the possibility of implementing each stage on a CPU. Fine–grained parallelization was also not followed, since four CPUs are not adequate for supporting such a

level of parallelism. Second, local memory limitations (64 Kbytes per CPU) on each node forced us to use an approach that did not require large data sets to be simultaneously resident within each local memory module. The implementation that we finally converged on is described below.

The Unix-based version basically used multiple programs, and piped them together. The DSP-based version for the 4-CPU prototype essentially takes these multiple programs, runs them on different processors, and uses the communication hardware to act as the pipes. However, since we only have 4 processors, it was not possible to map each program to a single dedicated processor. Nor was this necessary, for that matter -- due to the varying amount of computation needed by each program. Rather, the programs were "multiplexed" as needed on the given processors. This may have inhibited some amount of parallelism, but other choices were not available.

Our complexity analysis for the Speaker ID program (Section 3.2) had established that the *classify()* operation would take the most time, by far. One call to *classify* is needed for each feature, for a total of 4, thus far exceeding the complexity of all other routines, including the CEPS module. We split the classification in half, 2 on processor B and 2 on processor D. Processors A and C performed the feature extractions. The breakdown of activities for each CPU, which each CPU repeats cyclically, is as follows:

```
CPU_A:      read input samples (from a memory buffer);
            do I/O overlapping;
            do cepstrum;
            do first derivative of cepstrum;
            Send the 'ceps' and 'dceps' features to CPU_B and CPU_C, in an
            interleaved fashion through the communication channels;

CPU_B:      calculate windowed cepstrum (a single multiplication per sample);
            classify windowed ceps;
            classify dceps;
            at the very end, send classification results to CPU_D, which computes the
            final speaker ranking;

CPU_C:      calculate rasta;
            calculate second derivative;
            Send these two features to CPU_D for classification;

CPU_D:      classify rasta;
            classify dceps2;
            at the very end, accumulate all classification results, compute speaker
            rankings;
```

The communication pattern among the four CPUs are thus as shown in Figure 8.

Even though two types of data are sent over the lines from CPU_A TO CPU_B and FROM CPU_ A TO CPU_C, a single communication channel is used for both data types. (Two communication channels are available, but a bug in the emulator prevented their concurrent use.) The different features are simply interleaved (without any "message type" to indicate which feature is being sent; it's assumed that the two processors are synchronized using separate synchronization messages). The extracted tables and training data are all kept within the local memory of the appropriate CPUs.

An implementation of a DSP prototype with adequate I/O facilities will require CPU_A to control the A-D font end, while CPU_D will initiate transfers of the ranking data to the host console.

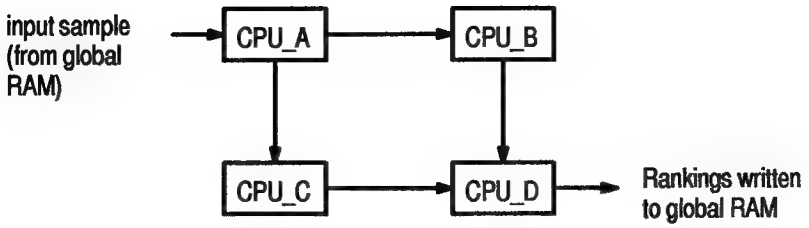


Figure 8. Communication patterns in the parallelized Spkrid code for the 4–CPU prototype.

3.4 Performance Of The Spkrid Code On The Prototype

The Parallel Processing Development System (PPDS) on which the prototype was based, unfortunately, did not include facilities to measure the overall execution time of the parallel implementation, with communication delays. However, it allowed the execution time of the code on each processor to be accurately measured in numbers of processor cycles. Since the amount of inter–CPU communications are sparse and overlapped with the computations, information gleaned from the individual cycle counts are expected to be quite close to the cycle counts for the overall implementation. In particular, the *worst–case* latency of a single pass of the Speaker ID program can be estimated as:

cycle count for CPU_A + max (cycle counts of CPU_B, CPU_C) + cycle count for CPU_D

We now present the cycle counts obtained from the profiler for the Speaker ID implementation for the DSP prototype for the recognition phase. We first present the cycle counts with *all* optimizations enabled in the compiler, and – for comparison, present the cycle counts for the unoptimized version. We finally present the cycle counts for the training phase.

The "inclusive" cycle counts includes the time spent in all functions that are called. The "exclusive" cycle counts reflect only the time spent in the code of the main function, and excludes the time spent within called functions. ("BP hits" refers to the number of hits to the branch predictor of each CPU.)

I. OPTIMIZED PARALLEL VERSION OF THE SPKRID PROGRAM: Cycle Counts

a. Cycle counts for CPU_a: optimized

```
Program Name:    /usr2/speaker/jason/new/cpu_a.out
Start Address:   40000000 main, at line 38, "/usr2/speaker/jason/new/cpu_a.c
Stop Address:    40000389          exit
Run Cycles:      1540988
Profile Cycles:  1540988
BP Hits:         2451


************************************************************************
     Area Name                Count   Inclusive    Incl-Max  Exclusive    Excl-Max
CF main()                       1     1540981     1540981      15209       15209
CF read_samples()               3       96012       53284      96012       53284
CF io_overlap()                 2      173574       97865     173574       97865
CF get_cepstrum()               2     1188216      660114    1177335      654069
```

```
CF derivative()                  30       50691        1750       45576        1529
CF get_dceps()                    3       61121       31854        9830        6069

*------------------------------------------------------------------------*
     Area Name                 Count
CF derivative()                  30  73% ====================================
CF get_dceps()                    3   7% ===
CF read_samples()                 3   7% ===
CF get_cepstrum()                 2   4% ==
CF io_overlap()                   2   4% ==
CF main()                         1   2% =

*------------------------------------------------------------------------*
     Area Name               Inclusive
CF main()                    1540981  99% ====================================
CF get_cepstrum()            1188216  77% ==============================
CF io_overlap()               173574  11% ====
CF read_samples()              96012   6% ==
CF get_dceps()                 61121   3% =
CF derivative()                50691   3% =

*------------------------------------------------------------------------*
     Area Name               Incl-Max
CF main()                    1540981  99% ====================================
CF get_cepstrum()             660114  42% ================
CF io_overlap()                97865   6% ==
CF read_samples()              53284   3% =
CF get_dceps()                 31854   2%
CF derivative()                 1750  <1%

*------------------------------------------------------------------------*
     Area Name               Exclusive
CF get_cepstrum()            1177335  76% ====================================
CF io_overlap()               173574  11% =====
CF read_samples()              96012   6% ===
CF derivative()                45576   2% =
CF main()                      15209  <1%
CF get_dceps()                  9830  <1%

*------------------------------------------------------------------------*
     Area Name               Excl-Max
CF get_cepstrum()             654069  42% ====================================
CF io_overlap()                97865   6% =====
CF read_samples()              53284   3% ===
CF main()                      15209  <1%
CF get_dceps()                  6069  <1%
CF derivative()                 1529  <1%

*------------------------------------------------------------------------*
     Area Name                Address
CF main()                    40000000
CF read_samples()            40000112
CF io_overlap()              40000133
CF get_cepstrum()            4000016c
CF derivative()              40000247
CF get_dceps()               40000268
```

b. Cycle counts for CPU_b: optimized

```
Program Name:   /usr2/speaker/jason/new/cpu_b.out
Start Address:  40000000 main, at line 43, "/usr2/speaker/jason/new/cpu_b.c
Stop Address:   400001e0          exit
```

```
Run Cycles:     4466203
Profile Cycles: 4466203
BP Hits:        909

*****************************************************************************
     Area Name                Count  Inclusive   Incl-Max  Exclusive   Excl-Max
CF main()                       1    4466196    4466196      43827      43827
CF classify()                   5    4405775    1223812    4405775    1223812

*****************************************************************************
     Area Name                Count
CF classify()                   5  83% ========================================
CF main()                       1  16% =======

*****************************************************************************
     Area Name            Inclusive
CF main()                 4466196  99% ========================================
CF classify()             4405775  98% =======================================

*****************************************************************************
     Area Name            Incl-Max
CF main()                 4466196  99% ========================================
CF classify()             1223812  27% ==========

*****************************************************************************
     Area Name            Exclusive
CF classify()             4405775  98% ========================================
CF main()                   43827  <1%

*****************************************************************************
     Area Name            Excl-Max
CF classify()             1223812  27% ========================================
CF main()                   43827  <1% =

*****************************************************************************
     Area Name             Address
CF main()                 40000000
CF classify()             400000fc
```

c. Cycle counts for CPU_c: optimized

```
Program Name:   /usr2/speaker/jason/new/cpu_c.out
Start Address:  40000000 main, at line 42, "/usr2/speaker/jason/new/cpu_c.c
Stop Address:   40000351          exit
Run Cycles:     151372
Profile Cycles: 151372
BP Hits:        819

*****************************************************************************
     Area Name                Count  Inclusive   Incl-Max  Exclusive   Excl-Max
CF main()                       1     151365     151365      48467      48467
CF get_rasta()                  2      34411      18740      34411      18740
CF derivative()                30      50691       1750      45576       1529
CF get_dceps()                  4      60767      25527       9476       4950


*****************************************************************************
     Area Name                Count
CF derivative()                30  81% ========================================
CF get_dceps()                  4  10% ====
CF get_rasta()                  2   5% ==
CF main()                       1   2% =
```

```
*****************************************************************************
     Area Name            Inclusive
CF main()                  151365  99% ====================================
CF get_dceps()              60767  40% ==============
CF derivative()             50691  33% ============
CF get_rasta()              34411  22% ========

*****************************************************************************
     Area Name            Incl-Max
CF main()                  151365  99% ====================================
CF get_dceps()              25527  16% ======
CF get_rasta()              18740  12% ====
CF derivative()              1750   1%

*****************************************************************************
     Area Name            Exclusive
CF main()                   48467  32% ================================
CF derivative()             45576  30% ===============================
CF get_rasta()              34411  22% ========================
CF get_dceps()               9476   6% =======

*****************************************************************************
     Area Name            Excl-Max
CF main()                   48467  32% ================================
CF get_rasta()              18740  12% ============
CF get_dceps()               4950   3% ===
CF derivative()              1529   1% =

*****************************************************************************
     Area Name             Address
CF main()                 40000000
CF get_rasta()            40000113
CF derivative()           400001c4
CF get_dceps()            400001e5
```

d. Cycle counts for CPU_d: optimized

```
Program Name:    /usr2/speaker/jason/new/cpu_d.out
Start Address:   40000000 main, at line 52, "/usr2/speaker/jason/new/cpu_d.c
Stop Address:    400001da          exit
Run Cycles:      4455040
Profile Cycles:  4455040
BP Hits:         162

*****************************************************************************
     Area Name              Count  Inclusive   Incl-Max  Exclusive   Excl-Max
CF main()                       1    4455033    4455033      44252      44252
CF classify()                   6    4405812    1223812    4405812    1223812

*****************************************************************************
     Area Name              Count
CF classify()                   6  85% ====================================
CF main()                       1  14% ======

*****************************************************************************
     Area Name            Inclusive
CF main()                  4455033  99% ====================================
CF classify()              4405812  98% ===================================

*****************************************************************************
     Area Name            Incl-Max
CF main()                  4455033  99% ====================================
CF classify()              1223812  27% ==========
```

```
*****************************************************************************
     Area Name            Exclusive
CF classify()              4405812  98% ====================================
CF main()                    44252  <1%

*****************************************************************************
     Area Name             Excl-Max
CF classify()              1223812  27% ====================================
CF main()                    44252  <1% =

*****************************************************************************
     Area Name              Address
CF main()                  40000000
CF classify()              4000011e
```

II. UN-OPTIMIZED PARALLEL VERSION OF THE SPKRID PROGRAM: Cycle Counts

– Included only for reference, to indicate cycle count improvements through optimizations.

a. Cycle counts for CPU_a: unoptimized

```
Program Name:    /usr2/speaker/jason/new/cpu_a.out
Start Address:   4000007c main, at line 38, "/usr2/speaker/jason/new/cpu_a.c
Stop Address:    40000444          exit
Run Cycles:      2879131
Profile Cycles:  2879131
BP Hits:         2725

*****************************************************************************
     Area Name                 Count  Inclusive   Incl-Max  Exclusive   Excl-Max
CF ClearMsg()                      0          0          0          0          0
CF WaitFor()                      14       6439        876       2005        277
CF CreateMsg()                     5      12957       3591      12957       3591
CF SendMsg()                      10       8164       1089        400         40
CF RecvMsg()                       0          0          0          0          0
CF EndMsg()                        4       1194        301        132         33
CF main()                          1    2879124    2879124       2935       2935
CF read_samples()                  3     118125      65560     118125      65560
CF io_overlap()                    2     283065     159553     283065     159553
CF power()                        27     111051       4113     111051       4113
CF auto_corr()                    27    1795716      66508    1795419      66497
CF lpca()                         27     123957       4591     119799       4437
CF lpc_cepstrum()                 27      88884       3292      82458       3054
CF get_cepstrum()                  2    2344986    1302768     225378     125208
CF derivative()                   30      89481       3043      84366       2822
CF get_dceps()                     3     107698      55294      17617      10114

*****************************************************************************
     Area Name                 Count
CF derivative()                   30  16% ====================================
CF auto_corr()                    27  14% ==================================
CF lpc_cepstrum()                 27  14% ==================================
CF lpca()                         27  14% ==================================
CF power()                        27  14% ==================================
CF WaitFor()                      14   7% =================
CF SendMsg()                      10   5% ============
CF CreateMsg()                     5   2% ======
CF EndMsg()                        4   2% ====
CF get_dceps()                     3   1% ===
CF read_samples()                  3   1% ===
CF get_cepstrum()                  2   1% ==
```

```
CF io_overlap()                2   1% ==
CF main()                      1  <1% =
CF ClearMsg()                  0   0%
CF RecvMsg()                   0   0%

********************************************************************************
     Area Name           Inclusive
CF main()                2879124  99% ========================================
CF get_cepstrum()        2344986  81% ================================
CF auto_corr()           1795716  62% ========================
CF io_overlap()           283065   9% ===
CF lpca()                 123957   4% =
CF read_samples()         118125   4% =
CF power()                111051   3% =
CF get_dceps()            107698   3% =
CF derivative()            89481   3% =
CF lpc_cepstrum()          88884   3% =
CF CreateMsg()             12957  <1%
CF SendMsg()                8164  <1%
CF WaitFor()                6439  <1%
CF EndMsg()                 1194  <1%
CF ClearMsg()                  0   0%
CF RecvMsg()                   0   0%

********************************************************************************
     Area Name            Incl-Max
CF main()                2879124  99% ========================================
CF get_cepstrum()        1302768  45% ================
CF io_overlap()           159553   5% ==
CF auto_corr()             66508   2%
CF read_samples()          65560   2%
CF get_dceps()             55294   1%
CF lpca()                   4591  <1%
CF power()                  4113  <1%
CF CreateMsg()              3591  <1%
CF lpc_cepstrum()           3292  <1%
CF derivative()             3043  <1%
CF SendMsg()                1089  <1%
CF WaitFor()                 876  <1%
CF EndMsg()                  301  <1%
CF ClearMsg()                  0   0%
CF RecvMsg()                   0   0%

********************************************************************************
     Area Name           Exclusive
CF auto_corr()           1795419  62% ========================================
CF io_overlap()           283065   9% =====
CF get_cepstrum()         225378   7% ====
CF lpca()                 119799   4% ==
CF read_samples()         118125   4% ==
CF power()                111051   3% ==
CF derivative()            84366   2% =
CF lpc_cepstrum()          82458   2% =
CF get_dceps()             17617  <1%
CF CreateMsg()             12957  <1%
CF main()                   2935  <1%
CF WaitFor()                2005  <1%
CF SendMsg()                 400  <1%
CF EndMsg()                  132  <1%
CF ClearMsg()                  0   0%
CF RecvMsg()                   0   0%
```

```
*************************************************************************
     Area Name              Excl-Max
CF io_overlap()              159553   5% ====================================
CF get_cepstrum()            125208   4% ============================
CF auto_corr()                66497   2% ===============
CF read_samples()             65560   2% ===============
CF get_dceps()                10114  <1% ==
CF lpca()                      4437  <1% =
CF power()                     4113  <1%
CF CreateMsg()                 3591  <1%
CF lpc_cepstrum()              3054  <1%
CF main()                      2935  <1%
CF derivative()                2822  <1%
CF WaitFor()                    277  <1%
CF SendMsg()                     40  <1%
CF EndMsg()                      33  <1%
CF ClearMsg()                     0   0%
CF RecvMsg()                      0   0%

*************************************************************************
     Area Name               Address
CF ClearMsg()               40000000
CF WaitFor()                40000017
CF CreateMsg()              40000023
CF SendMsg()                4000003a
CF RecvMsg()                40000051
CF EndMsg()                 40000069
CF main()                   4000007c
CF read_samples()           40000135
CF io_overlap()             40000155
CF power()                  400001a5
CF auto_corr()              400001bb
CF lpca()                   400001ea
CF lpc_cepstrum()           4000023b
CF get_cepstrum()           40000277
CF derivative()             400002f1
CF get_dceps()              40000316
```

b. Cycle counts for CPU_b: unoptimized

```
Program Name:    /usr2/speaker/jason/new/cpu_b.out
Start Address:   4000007c main, at line 43, "/usr2/speaker/jason/new/cpu_b.c
Stop Address:    40000252          exit
Run Cycles:      14204212
Profile Cycles:  14204212
BP Hits:         1002

*************************************************************************
     Area Name              Count  Inclusive   Incl-Max  Exclusive   Excl-Max
CF ClearMsg()                   7      29953       4279      29953       4279
CF WaitFor()                    9       3766        894        966        206
CF CreateMsg()                  0          0          0          0          0
CF SendMsg()                    2        628        314         80         40
CF RecvMsg()                    7       5076       1110        308         44
CF EndMsg()                     0          0          0          0          0
CF main()                       1   14204205   14204205      18524      18524
CF classify()                   5   14137316    3927016   14137316    3927016

*************************************************************************
     Area Name              Count
CF WaitFor()                    9  29% ====================================
```

```
CF ClearMsg()                7  22% ============================
CF RecvMsg()                 7  22% ============================
CF classify()                5  16% ====================
CF SendMsg()                 2   6% ========
CF main()                    1   3% ====
CF CreateMsg()               0   0%
CF EndMsg()                  0   0%

*****************************************************************************
     Area Name            Inclusive
CF main()                14204205  99% ========================================
CF classify()            14137316  99% ========================================
CF ClearMsg()               29953  <1%
CF RecvMsg()                 5076  <1%
CF WaitFor()                 3766  <1%
CF SendMsg()                  628  <1%
CF CreateMsg()                  0   0%
CF EndMsg()                     0   0%

*****************************************************************************
     Area Name             Incl-Max
CF main()                14204205  99% ========================================
CF classify()             3927016  27% ==========
CF ClearMsg()                4279  <1%
CF RecvMsg()                 1110  <1%
CF WaitFor()                  894  <1%
CF SendMsg()                  314  <1%
CF CreateMsg()                  0   0%
CF EndMsg()                     0   0%

*****************************************************************************
     Area Name            Exclusive
CF classify()            14137316  99% ========================================
CF ClearMsg()               29953  <1%
CF main()                   18524  <1%
CF WaitFor()                  966  <1%
CF RecvMsg()                  308  <1%
CF SendMsg()                   80  <1%
CF CreateMsg()                  0   0%
CF EndMsg()                     0   0%

*****************************************************************************
     Area Name             Excl-Max
CF classify()             3927016  27% ==========
CF main()                   18524  <1%
CF ClearMsg()                4279  <1%
CF WaitFor()                  206  <1%
CF RecvMsg()                   44  <1%
CF SendMsg()                   40  <1%
CF CreateMsg()                  0   0%
CF EndMsg()                     0   0%

*****************************************************************************
     Area Name              Address
CF ClearMsg()            40000000
CF WaitFor()             40000017
CF CreateMsg()           40000023
CF SendMsg()             4000003a
CF RecvMsg()             40000051
CF EndMsg()              40000069
CF main()                4000007c
```

```
CF classify()            4000015c
```

c. Cycle counts for CPU_c: unoptimized

```
Program Name:   /usr2/speaker/jason/new/cpu_c.out
Start Address:  4000007c main, at line 42, "/usr2/speaker/jason/new/cpu_c.c
Stop Address:   400003d9          exit
Run Cycles:     234697
Profile Cycles: 234697
BP Hits:        952

*****************************************************************************
     Area Name                Count   Inclusive    Incl-Max  Exclusive    Excl-Max
CF ClearMsg()                     7       29953        4279       29953        4279
CF WaitFor()                     15        6601         895        1806         220
CF CreateMsg()                    6       12978        3591       12978        3591
CF SendMsg()                      6        4094         967         240          40
CF RecvMsg()                      7        5105        1115         308          44
CF EndMsg()                       2         592         296          66          33
CF main()                         1      234690      234690       11167       11167
CF rasta()                       28       28602        1140       28602        1140
CF get_rasta()                    2       63116       34147       34514       18187
CF derivative()                  30       89481        3043       84366        2822
CF get_dceps()                    4      107258       44278       17177        8185

*****************************************************************************
     Area Name               Count
CF derivative()                 30  27% ===========================================
CF rasta()                      28  25% ========================================
CF WaitFor()                    15  13% ====================
CF ClearMsg()                    7   6% ========
CF RecvMsg()                     7   6% ========
CF CreateMsg()                   6   5% =======
CF SendMsg()                     6   5% =======
CF get_dceps()                   4   3% ====
CF EndMsg()                      2   1% ==
CF get_rasta()                   2   1% ==
CF main()                        1  <1% =

*****************************************************************************
     Area Name            Inclusive
CF main()                   234690  99% ===========================================
CF get_dceps()              107258  45% ================
CF derivative()              89481  38% ==============
CF get_rasta()               63116  26% =========
CF ClearMsg()                29953  12% ====
CF rasta()                   28602  12% ====
CF CreateMsg()               12978   5% ==
CF WaitFor()                  6601   2% =
CF RecvMsg()                  5105   2%
CF SendMsg()                  4094   1%
CF EndMsg()                    592  <1%

*****************************************************************************
     Area Name             Incl-Max
CF main()                   234690  99% ===========================================
CF get_dceps()               44278  18% ======
CF get_rasta()               34147  14% =====
CF ClearMsg()                 4279   1%
CF CreateMsg()                3591   1%
CF derivative()               3043   1%
```

```
CF rasta()                    1140  <1%
CF RecvMsg()                  1115  <1%
CF SendMsg()                   967  <1%
CF WaitFor()                   895  <1%
CF EndMsg()                    296  <1%

*****************************************************************************
     Area Name            Exclusive
CF derivative()              84366  35% ====================================
CF get_rasta()               34514  14% ==============
CF ClearMsg()                29953  12% ============
CF rasta()                   28602  12% ============
CF get_dceps()               17177   7% =======
CF CreateMsg()               12978   5% =====
CF main()                    11167   4% ====
CF WaitFor()                  1806  <1%
CF RecvMsg()                   308  <1%
CF SendMsg()                   240  <1%
CF EndMsg()                     66  <1%

*****************************************************************************
     Area Name             Excl-Max
CF get_rasta()               18187   7% ===================================
CF main()                    11167   4% ======================
CF get_dceps()                8185   3% ================
CF ClearMsg()                 4279   1% ========
CF CreateMsg()                3591   1% =======
CF derivative()               2822   1% =====
CF rasta()                    1140  <1% ==
CF WaitFor()                   220  <1%
CF RecvMsg()                    44  <1%
CF SendMsg()                    40  <1%
CF EndMsg()                     33  <1%

*****************************************************************************
     Area Name              Address
CF ClearMsg()              40000000
CF WaitFor()               40000017
CF CreateMsg()             40000023
CF SendMsg()               4000003a
CF RecvMsg()               40000051
CF EndMsg()                40000069
CF main()                  4000007c
CF rasta()                 40000147
CF get_rasta()             400001b1
CF derivative()            4000023b
CF get_dceps()             40000260
```

d. Cycle counts for CPU_d: unoptimized

```
Program Name:    /usr2/speaker/jason/new/cpu_d.out
Start Address:   4000007c main, at line 52, "/usr2/speaker/jason/new/cpu_d.c
Stop Address:    40000259          exit
Run Cycles:      14192292
Profile Cycles:  14192292
BP Hits:         268

*****************************************************************************
     Area Name                Count  Inclusive   Incl-Max  Exclusive   Excl-Max
CF ClearMsg()                    10      42790       4279      42790       4279
CF WaitFor()                     10       3894        894        980        206
CF CreateMsg()                    0          0          0          0          0
```

```
CF SendMsg()                0          0          0          0          0
CF RecvMsg()               10       6062       1110        440         44
CF EndMsg()                 0          0          0          0          0
CF main()                   1   14192285   14192285       5511       5511
CF classify()               6   14136977    3926856   14136977    3926856

*****************************************************************************
     Area Name              Count
CF ClearMsg()                 10  27% ==========================================
CF RecvMsg()                  10  27% ==========================================
CF WaitFor()                  10  27% ==========================================
CF classify()                  6  16% ========================
CF main()                      1   2% ===
CF CreateMsg()                 0   0%
CF EndMsg()                    0   0%
CF SendMsg()                   0   0%

*****************************************************************************
     Area Name            Inclusive
CF main()                 14192285  99% ======================================
CF classify()             14136977  99% ======================================
CF ClearMsg()                42790  <1%
CF RecvMsg()                  6062  <1%
CF WaitFor()                  3894  <1%
CF CreateMsg()                   0   0%
CF EndMsg()                      0   0%
CF SendMsg()                     0   0%

*****************************************************************************
     Area Name             Incl-Max
CF main()                 14192285  99% ======================================
CF classify()              3926856  27% ==========
CF ClearMsg()                 4279  <1%
CF RecvMsg()                  1110  <1%
CF WaitFor()                   894  <1%
CF CreateMsg()                   0   0%
CF EndMsg()                      0   0%
CF SendMsg()                     0   0%

*****************************************************************************
     Area Name            Exclusive
CF classify()             14136977  99% ======================================
CF ClearMsg()                42790  <1%
CF main()                     5511  <1%
CF WaitFor()                   980  <1%
CF RecvMsg()                   440  <1%
CF CreateMsg()                   0   0%
CF EndMsg()                      0   0%
CF SendMsg()                     0   0%

*****************************************************************************
     Area Name             Excl-Max
CF classify()              3926856  27% ======================================
CF main()                     5511  <1%
CF ClearMsg()                 4279  <1%
CF WaitFor()                   206  <1%
CF RecvMsg()                    44  <1%
CF CreateMsg()                   0   0%
CF EndMsg()                      0   0%
CF SendMsg()                     0   0%

*****************************************************************************
     Area Name             Address
```

```
CF ClearMsg()           40000000
CF WaitFor()            40000017
CF CreateMsg()          40000023
CF SendMsg()            4000003a
CF RecvMsg()            40000051
CF EndMsg()             40000069
CF main()               4000007c
CF classify()           4000018b
```

III. TRAINING OVERHEAD CYCLE COUNTS (ONLY CPU_B IS INVOLVED)

```
Program Name:   /usr2/speaker/jason/train/cpu_b.out
Start Address:  40000000 main, at line 44, "/usr2/speaker/jason/train/cpu_b
Stop Address:   40000506          exit
Run Cycles:     215866557
Profile Cycles: 215866557
BP Hits:        1455137

*************************************************************************
     Area Name                Count  Inclusive    Incl-Max  Exclusive   Excl-Max
CF find_mse()                351417   69931983         199   66066396        188
CF indexx()                   11638   62507186        8656   62507186       8656
CF train()                        2  202200462   114481606     627204     313625
CF lloyd()                       26  201366948    18251509   68822399    5026066
CF ran3()                      2184     197890       16276     182180       8421

*************************************************************************
     Area Name                Count
CF find_mse()                351417  96% ======================================
CF indexx()                   11638   3% =
CF ran3()                      2184  <1%
CF lloyd()                       26  <1%
CF train()                        2  <1%

*************************************************************************
     Area Name             Inclusive
CF train()                202200462  93% ======================================
CF lloyd()                201366948  93% ======================================
CF find_mse()              69931983  32% =============
CF indexx()                62507186  28% ============
CF ran3()                    197890  <1%

*************************************************************************
     Area Name              Incl-Max
CF train()                114481606  53% ======================================
CF lloyd()                 18251509   8% =====
CF ran3()                    16276  <1%
CF indexx()                   8656  <1%
CF find_mse()                  199  <1%

*************************************************************************
     Area Name             Exclusive
CF lloyd()                 68822399  31% ========================================
CF find_mse()              66066396  30% =======================================
CF indexx()                62507186  28% ====================================
CF train()                   627204  <1%
CF ran3()                    182180  <1%

*************************************************************************
     Area Name              Excl-Max
CF lloyd()                  5026066   2% ========================================
```

```
CF train()                  313625  <1% ==
CF indexx()                   8656  <1%
CF ran3()                     8421  <1%
CF find_mse()                  188  <1%

*****************************************************************************
     Area Name                Address
CF find_mse()               400000f1
CF indexx()                 4000010f
CF train()                  40000186
CF lloyd()                  400002c4
CF ran3()                   400003a7
```

3.5 Final Comments on the Performance Of The Spkrid Code On The Prototype

The performance results for the spkrid code on the prototype system suggests that the CPUs B and D are the most heavily loaded. Both of these CPUs spend the bulk of their cycles in the *classify*() routine. Note that the profiler has no way of measuring the overall execution time, since it cannot time the communication delays, and is limited to profiling one CPU at a time. We can, however, obtain a worst-case cycle count for the overall spkrid recognition phase by adding up the cycle counts of the individual CPUs. For such a worst-case scenario, the total cycle count for the recognition phase in the optimized version is:

1540988 (CPU_a) + 4466203 (CPU_b) + 151372 (CPU_c) + 4455040 (CPU_d) = 10613608 cycles.

For the 32 MHz. CPUs, this translates to a delay of roughly 301 milliseconds. Clearly, a reduction of this delay can be achieved by parallelization of the *classify* routine itself, requiring one or two additional CPUs. Further gains - possibly substantial - are also possible through the overlapping of computation and communication among the CPUs.

4. THE IMPLEMENTATION OF THE SPEECH ENHANCEMENT CODE ON THE PROTOTYPE SYSTEM

This section describes our approach to the porting and parallelization of the RL SEU program suite and an evaluations of the performance of the SEU code on the 4-CPU prototype system. In particular, we describe the important changes made to the code to facilitate the implementation on the prototype without compromising any functionality. The performance results reported here also identify the main cycle sinks in the program in an effort to show where parallelization efforts should be directed if additional CPUs are available for further performance enhancements.

4.1 The RL Speech Enhancement Program (SEU)

The Rome Labs SEU program suites are described in [WeAs 78, RLTR 92a, RLTR 92b], and we will thus not describe them in any depth in this report. Basically, the SEU program suites that were supplied allow:

a) Impulse noise to be detected and attenuated - i.e., impulse noise to be removed (the *Imp* program).

b) Tonal noise to be extracted and removed (the *HiTones* program).

c) Wideband random noise to be removed (the *Intel95* program).

In the code supplied, each program was self-contained and generally had its own complement of signal I/O functions.

4.2 Porting of SEU Programs to the Prototype System

The SEU95 source code for each speech enhancement functions *Imp*, *Hi_Tone* and *Intel95* (hereafter called SEU programs) were ported to the 4-CPU based prototype, each to run on a single CPU. The main reason to run one function per CPU was to allow a pipelined processing system to be set up, where these functions can be applied in sequence to the speech sample to be cleaned up. Thus, the porting of these functions to the 'C40 code required us to address the following differences between the Unix environment (where the supplied SEU95 code was run) and the environment for the 4-CPU prototype:

(a) Output function calls, such as printf, usable in the Unix version, were replaced by stores into the global memory of the prototype system.

(b) Input function calls (such as scanf()) that are usable in the Unix version were replaced by literal data (or by accessing data pre-stored in memory), since no explicit I/O function calls are supported in the prototype.

(c) Functions in the supplied SEU95 code that used large arrays as *local* variables were modified to ensure that the activation stack for the 'C40 code was confined to the CPU-internal RAM. (Confining the activation stack to the CPU-internal RAM allows the code to execute more efficiently. The default memory map for the 'C40s limit the activation stack size to 1 K words for this reason.)

(d) Heap management inefficiencies in the 'C40 environment were addressed to avoid repeated allocations and deallocations from the heap. Generally speaking, it is probably a good idea to *avoid* using dynamically allocated storage for real-time speech processing systems like the prototype, given that it isn't easy to detect (externally) when such memory runs out. It is also not clear as to what would be an universal way of coping with heap overflows in such systems. Thus, when the size of a variable is known at compile-time, it should be made static. If the maximum size of a variable is known, it should be allocated as such, so long as the size isn't too big. Doing so aids in determining the required amount of memory for a given program, and certainly improves the performance (no calls to *malloc/free*, no heap fragmentation and associated compaction delays).

The specific changes made to the SEU95 functions to get them running on the prototype are summarized below:

- As in the speaker ID port, the input sample was converted to an object file and linked into the executable, due to the inability of the 'C40 emulator to provide fast communication between the SPARC host and the prototype board. It is expected, obviously, that this change is temporary and these changes will not be needed when an appropriate A/D board will be available for the prototype.

- Any printf() calls or other I/O calls were removed or otherwise worked around, given that the 'C40 library has no such functions. For outputs, this was easy enough (most of the printf() calls were for error or warning messages). However, the SEU programs used a "command file" for input parameters. This file was replaced with hard-coded values. The values were the same as supplied in the test command files. Again, a fast communication facility allowing digital I/O between the SPARC host and the prototype board will allow us to download the parameter values directly into the RAM of the prototype, obviating the need to hard code the parameters.

- Similarly, the output from the SEU programs is also stored in memory. Specifically, the global (shared) memory was used for input and output. This memory is limited to 128 kilowords, thus limiting the size of test samples to around 64 kilowords.

- A small number of functions had large variables declared locally, i.e., on the program activation stack. Since the 'C40 default memory map only allows for a 1 kiloword stack, these variables would not fit on the stack. They were moved outside of the function, and made global.

- Our experience with the 'C40 environment suggests that the heap management functions in the 'C40 C library don't deal with fragmentation all that well. The main function for each SEU process dynamically allocates a number of variables, the sizes of which do not appear to change. Modifications were made such that these variables are only allocated once (the first call).

These changes are documented in-line in the 'C40 code, included in the appendix.

4.3 Performance of the SEU Programs on the Prototype

Due to reasons identical to that listed in Section 3.4, we could only measure the cycle counts for each CPU individually using the profiler. We now present these cycle counts for the SEU programs.

I. CYCLE COUNTS FOR THE PORTED SEU PROGRAMS: OPTIMIZED VERSION

The ported versions of the SEU95 programs *Imp*, *Hi_Tone* and *Intel95* were run on the prototype, one function per CPU, using the parameters supplied for each program in the sample inputs for the SEU95 code as given in the "command files". Each such run was profiled to estimate the cycle counts in various C functions within these SEU programs. The input used for all tests was 50000 samples in length.

Note that each program typically had a number of configuration options, which were given (originally) in command files. It is likely that a different set of options might result in significantly different cycle counts. Note also that profiling may inflate the overall number of cycles executed. Counts of "leaf" functions, those that make few subroutine calls themselves, should be fairly accurate.

We now present the results of profiling runs of each SEU program:

SEU Program: Imp

```
Overall cycles: 16,614,002

Function: ImpulseModulationProcess()
  number of samples processed per call ("impIOFrameSize"): 256
  total cycles (inclusive): 9,703,071
               (exclusive): 6,663,576
  number of calls: 196
  max cycles/call (inclusive): 52,776
                  (exclusive): 37,221
```

Summary: The *Imp* program spends roughly 193 cycles/call/sample. (As noted earlier, changes to the frame size and/or overlap percentage could have a profound effect on these numbers.) Of all the SEU programs, *Imp* requires the least processing time. Further speedup of this program through parallelization is thus unnecessary.

SEU Program: HiTone

```
Overall cycles: 120,104,860
```

```
Function: AttenuateHiTones()
  number of samples processed per call ("htIOFrameSize"): 512
  total cycles (inclusive): 112,972,161
               (exclusive):  15,333,218
  number of calls: 97
  max cycles/call (inclusive): 1,177,451
                  (exclusive):   164,162
```

The "inclusive" cycle counts includes the time spent in all functions that are called. The "exclusive" cycle counts reflect only the time spent in the code of the main function, and excludes the time spent within called functions.

Summary: The *HiTone* program spends roughly 2274 cycles/call/sample. (As noted earlier, changes to the frame size and/or overlap percentage could have a profound effect on these numbers.) The large difference between inclusive and exclusive cycle counts indicate that most of the processing is done in other functions called by *HiTone*. The profiling data for the major functions called within the body of *HiTone* are given below:

- The *Histogram* function is called by *AttenuateHiTones()* at most once per frame. Its cycle count is negligible, compared the cycle count for the *fftd()* function, as given below.

```
Function: Histogram()
  total cycles (inclusive): 1,994,141
               (exclusive): 1,992,942
  number of calls: 97
  max cycles/call (inclusive): 33,411
                  (exclusive): 33,400
```

- The *fftd* function is called twice per call to *AttenuateHiTones()*. This is the most cycle-intensive call:

```
Function: fftd()
  total cycles (inclusive): 75,826,054
               (exclusive): 66,360,298
  number of calls: 194  (twice per call to AttenuateHiTones())
  max cycles/call (inclusive): 400,598
                  (exclusive): 351,799
```

Thus, a clear way to speed up the *HiTone* program is to speed up the *fftd* function. There are several ways of doing this – all involving code parallelization – but, unfortunately, additional CPUs are not available in the prototype to support the parallelized *HiTone* program, as well as the other SEU programs and, possibly, a CPU devoted to I/O.

SEU Program: Intel95

```
Overall cycles: 206,412,821

Function: Intel95()
  number of samples processed per call ("intel95IOFrameSize"): 256
  total cycles (inclusive): 199,610,302
               (exclusive):  27,729,330
  number of calls: 194
  max cycles/call (inclusive): 1,031,677
                  (exclusive):   142,953
```

Summary: The *Intel95* program spends roughly 4019 cycles/call/sample. (Again, as noted earlier, changes to the frame size and/or overlap percentage could have a profound effect on these numbers.) As in

the case of *HiTone*, the large difference between inclusive and exclusive cycle counts indicate that most of the processing is done in other functions called by *Intel95*. Again, the bulk of the execution time of *Intel95* is spent within the function *fftd*. (Four calls are made to *fftd* by *Intel95*.) The profiling data for the *fftd* function called within the body of *fftd* are given below:

```
Function: fftd()
  total cycles (inclusive): 141,464,519
               (exclusive): 122,614,575
  number of calls: 4 in Intel95()
  max cycles/call (inclusive): 187,177
                  (exclusive): 162,880
```

Note that compared to *HiTone*, the frame size for *Intel95* is halved; The time spent *per call* to the *fftd* function in *Intel95* is less than half that the time per call to *fftd* in *HiTone*.

4.4 Final Comments on the Performance of SEU95 on the Prototype:

If all three of the SEU programs are implemented as a pipeline on the prototype, the CPU running the *Intel95* program will clearly be the bottleneck, being the slowest stage in the pipeline. One very simple way of surmounting this bottleneck will be to use two CPUs to implement *Intel95* – for example, by splitting up the 4 calls to *fftd* across the two CPUs. Assuming that such a parallelization halves the cycles spent in the calls to *fftd*, the overall cycle count per call per sample for *Intel95* reduces to roughly 2700 cycles. Note that even with this parallelization, *Intel95* is still the bottleneck. The pipeline processing rate that results from the implementation of *Intel95* on two CPUs is thus 2700 cycles per sample, assuming that inter–CPU communications are perfectly overlapped with the computations. For the 32 MHz. CPUs used in the prototype, this reduces to a pipeline cycle time of 81 microseconds. This means that the samples must not arrive at a rate higher than one per 81 microseconds, suggesting a peak sampling frequency of about 12.5 KHz. The approach just described was not implementable on the 4–CPU prototype, since it requires all 4 CPUs to be devoted to the computations, leaving no free CPU for interfacing to the A/D input and the D/A output.

5. LESSONS LEARNED

The performance results for the Spkrid code and the SEU code does shed some light on ways to further improve the latencies of these two programs. In particular, the following lessons were learned about the nature of the Spkrid and the SEU code and the system as a whole:

- It was relatively easy to transform, modify and code the applications – all at the source C code level – once a scheme for parallelizing the code had been devised.
- The main bottleneck for the speech enhancement system, running *Imp*, *HiTones* and *Intel95* in a pipelined configuration is the *Intel95* program. The long latency of the *Intel95* program is attributed to the long execution time of the *fftd* routine. Further performance gains can be realized by parallelizing the calls to the *fftd* routine and/or parallelizing the *fftd* routine itself.
- For the speaker id code, the main bottleneck is the *classify* routine. The only serious solution for further performance improvement in this case can come from parallelization of the body of this code. Fortunately, the loop level parallelism within *classify* is adequate for this purpose.
- The addition of just two more CPUs to the prototype cam greatly improve the usability and performance of the system for the applications that we implemented.

- The basic parallel processor development system (PPDS) hardware used for the prototype is not a good choice for implementing a fully functional system, complete with host, A/D, D/A interfaces. This is mainly due to the lack of a high speed connection between the host and the PPDS, as well as due to the inability of the PPDS to accommodate standard A/D, D/A cards. The recently available DSP multiprocessing boards based on the TMS 320C40 are better candidates for a real-time, high-end speech processing multiprocessor.

6. CONCLUSIONS

The implementation of the prototype system, including the parallel implementation of the spkrid code and the SEU programs and the performance of these applications on the prototype indicates that the basic goals of the project were met. However, the limitation of the prototype to just 4 CPUs precluded the incorporation of an A/D, D/A interface that operated in real-time. The implementation of the Spkrid code, as described, has a latency of 300 milliseconds (on the given datasets, with 14 speakers and 4 features). The code tables are small enough to be stored within the on-chip RAM of the CPUs, allowing for some performance gains. The SEU programs can all be implemented in a pipelined chain, and the resulting system allows for a sampling frequency up to 12 KHz. This is beyond the currently-used sampling frequency of 10 KHz. A more usable system should incorporate at least two more CPUs to allow front and back-end functions to be implemented.

REFERENCES

[WeAs 78] Weiss, Mark R. and Aschkenasy, Ernest, "The Speech Enhancement Advanced Development Model", Rome Lab Technical report RADC-TR-78-232, November 1978.

[RLTR 92a] "Noise Reduction Improvements", Interim Report on Contract No. F30602-87-0131, Rome Labs, December 1992.

[RLTR 92b] "Noise Reduction Improvements", Final Report on Contract No. f30602-87-0131, Rome Labs, December 1992.

*U.S. GOVERNMENT PRINTING OFFICE: 1997-610-130-61083

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.